Cost-effective Resource Provisioning for Spark Workloads

Yuxing Chen[†], Jiaheng Lu[†], Chen Chen[‡], Mohammad Hoque[†], Sasu Tarkoma[†]
[†]University of Helsinki, {yuxing.chen, jiaheng.lu, mohammad.a.hoque, sasu.tarkoma}@helsinki.fi
[‡]Huawei Canada Research Centre, chen.cc@huawei.com

ABSTRACT

Spark is one of the prevalent big data analytical platforms. Configuring proper resource provision for Spark jobs is challenging but essential for organizations to save time, achieve high resource utilization, and remain cost-effective. In this paper, we study the challenge of determining the proper parameter values that meet the performance requirements of workloads while minimizing both resource cost and resource utilization time. We propose a simulation-based cost model to predict the performance of jobs accurately. We achieve low-cost training by taking advantage of *simulation framework*, i.e., Monte Carlo (MC) simulation, which uses a small amount of data and resources to make a reliable prediction for larger datasets and clusters. The salient feature of our method is that it allows us to invest low training cost while obtaining an accurate prediction. Through experiments with six benchmark workloads, we demonstrate that the cost model yields less than 7% error on average prediction accuracy and the recommendation achieves up to 5x resource cost saving.

CCS CONCEPTS

• **Computing methodologies** → *Modeling methodologies.*

KEYWORDS

Resource provisioning; Spark executor parameter; Simulation; Cost model; Performance metrics.

ACM Reference Format:
Yuxing Chen, Jiaheng Lu, Chen Chen, Mohammad Hoque, and Sasu Tarkoma. 2019. Cost-effective Resource Provisioning for Spark Workloads. In *The 28th ACM International Conference on Information and Knowledge Management (CIKM '19), November 3–7, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3357384.3358090

1 INTRODUCTION

Modern big data platforms enable data scientists, predictive modelers, and statisticians to analyze, manage, and process big data. These platforms utilize resources from a local cluster or a cloud service provider, and the organizations are putting more emphasis on running jobs cost-effectively ([8, 9]). Spark [14] plays an important role in big data analytics academically and industrially. Yet, configuring proper resource portions for Spark jobs remains challenging. This paper addresses the challenge of provisioning resources for the Spark jobs in a cluster towards cost-effectiveness.

The existing research works model the performance of Spark workloads through machine learning techniques (e.g, [5, 12]) and various cost models (e.g., [11, 13]). In machine Learning methods, *Wang et al.* [12] investigated several common binary and multi-classification algorithms and found the decision tree (C5.0) to be the best. *Hernández et al.* [5] utilized regression models to predict the Spark job completion time based on a set of metrics at both system and application level. However, they relied heavily on historical logs and spent a great amount of time on model training. *Wang et al.* [13] and Ernest [11] proposed to execute the workloads on smaller input sizes and used cost functions to extrapolate the performance for larger input sizes. They neither considered the bottlenecks of network and disk I/O nor recommended parameter values.

In this paper, we develop a simulation-based framework with a linear cost model which considers both the network and disk I/O bottlenecks. The framework chooses the simulation runs adaptively, making the prediction efficient and reliable. The contributions of this paper are summarized as follows:

Low-cost simulation-based prediction model. We propose a simulation-based cost model to predict the performance of jobs accurately. We achieve low-cost training by taking advantage of a *simulation framework*, i.e., Monte Carlo (MC) simulation [1], which uses a small amount of data and resources to make a reliable prediction for larger datasets and clusters.

Cost-effective parameter recommendation. We introduce *resource-time* and the recommendation model to find cost-effective configurations via a range search algorithm. Assigning all available resources to a job does not necessarily lead to a better performance concerning resource utilization or resource time.

Comprehensive experiments. We ran experiments with six HiBench [6] workloads. In our implementation, a production environment requires only a small amount of the input data for training. As fewer resource and data are required, the training cost is considerably low (average 5.71% and 12.88% of total data size in terms of 80% and 90% *confident interval*, respectively). Experiments demonstrate that our model yields less than 7% error on average prediction accuracy and achieves cost-effective recommendation on jobs' performance (up to 5x gain in terms of resource cost).

2 PROBLEM STATEMENT

We define the research problem as follows:

- **Problem input:** (i) a set of new coming jobs $\mathbb{J}$ (each job $\mathcal{J}$ with data size D); (ii) resource profiles (cluster resource information, i.e., available vcore and memory, number of nodes); (iii) user-defined requirements (e.g., deadline T).
- **Problem output:** for each job $\mathcal{J}$, the output is resource profiles of (i) number of vcores V; (ii) amount of memory M

CIKM '19, November 3–7, 2019, Beijing, China

ACM ISBN 978-1-4503-6976-3/19/11…$15.00
https://doi.org/10.1145/3357384.3358090

- **Goal:** The goal is to recommend V and M towards cost-effective performance.

The essence of the goal is to provision resource ([2, 3, 10]) cost-effectively. To achieve this goal, we build a model for runtime prediction by executing the jobs with a small amount of input data. We introduce the resource-time (e.g., used in [4]) metric which quantifies the resource used and the duration of use. Given the job execution time T and the allocated vcore V and memory M, we define resource-time:

$$T_R = V \times T + \theta \times M \times T \tag{1}$$

where θ is the ratio of per GB memory price and per vcore price. We achieve the cost-effective vcore and memory parameter values to recommend by minimizing T_R:

$$(v_{rec}, m_{rec}) = arg \min_{v \in V, m \in M} T_R(v, m) \tag{2}$$

3 APPROACH

We first sample the input data for a job based on the data size and available resources and simulate the job with the sampled data for prediction. Second, we adaptively repeat simulations to identify the assured memory amount, thus to predict the memory for original data size. Thirdly, we analyze the characteristics of jobs with sampled data and build the cost model to predict the vcore performance of the job with larger input data size and resource amount. Finally, we target to optimize resource time.

Simulation. We simulate a job's performance with a small fraction of input data and then extrapolate the performance with the actual data size. We employ *Independent Bernoulli Sampling* for sampling input data, assuming an independent and identical distribution. We execute a job with the sample input data to predict the actual runtime by a Cost Model (denoted by C_m), i.e., $\hat{T} = Cost(\hat{X}, V, N)$, where $\hat{X}$, V, and N stand for data size, vcore, and number of physical machines. It is a challenge to determine the sample size. In particular, a sample set may be too small to represent the characteristics of the job with actual data in the following two aspects:

(1) **Data skewness:** the runtime of a job can vary with two different input data samples having the same size.
(2) **Runtime deviation:** the runtime of a job can vary with the same data and resource configuration.

To mitigate these two problems, we adopt Monte Carlo (MC) Simulation [1] with *replacement sampling*.

The cost model can be transformed into a linear form, i.e. $T = Cost(D, V, N) = \alpha D + \beta$ given V and N. So we estimate runtime $\hat{T} = \alpha \hat{X} + \beta$ given sample data $\hat{X}$. The mean runtime $\hat{\mu}_{\hat{T}}$ converges to a normal distribution as simulation n approaches infinity:

$$\hat{\mu}_{\hat{T}} \sim \mathcal{N}(\alpha E[\hat{X} + \beta], Var[\alpha^2 \hat{X}]/n) \;\; i.e., \;\; \mathcal{N}(E[\hat{T}], Var[\hat{T}]/n) \tag{3}$$

Equation 3 provides the *confidence interval* (CI) of cost model $CI(C_m)=[L_{C_m}, U_{C_m}]=[\hat{\mu}_{C_m} - t_* \cdot \hat{\sigma}_{C_m}/\sqrt{n}, \hat{\mu}_{C_m} + t_* \cdot \hat{\sigma}_{C_m}/\sqrt{n}]$. Given the estimated $\{\hat{T}_1, \hat{T}_2, ..., \hat{T}_n\}$ and the mean $\bar{T}$ from n sample simulation, the variance formulation is $\hat{\sigma}^2_{C_m} = \frac{1}{n-1} \sum_{i=1}^{n} (\hat{T}_i - \bar{T})^2$. The simulation terminates when the given confidence level holds $t_* \, \hat{\sigma}_{C_m}/\sqrt{n} < \epsilon$. The training cost accumulates T_R of sample simulations. The ratio of T_R for simulations and actual run can be expressed:

$$C_{training} = \frac{\sum_{i=1}^{n} T_{r_i}}{T_R} \tag{4}$$

where T_{r_i} and T_R are the resource time of i^{th} simulation and actual run; n is the number of simulations. Intuitively, a higher confidence implies a better prediction accuracy but more simulations.

Memory Assurance. The simulation results for memory resemble an elbow curve [7]. We consider the elbow point where the derivative of the curve function has a huge jump to discover the near-optimal memory size for the best resource time. Let P=$\{p_1, ...p_i, ..., p_n\}$ is a set of distinct sampling probabilities, where p_i is the i-th sample's probability. Let m_i denote the assured memory (determined by the elbow method) for each p_i, and M=$\{m_1, ...m_i, ..., m_n\}$. Now, we can predict the assured memory $\hat{M}$ for the total input data by a regression model given P and M. We employ a *linear regression* model, as it performs empirically well for our benchmark workloads.

Performance Prediction. We formally define the cost model. Let $\mathcal{J}$, D, V, and N be the job, input data size, number of vcores, and number of nodes, respectively. Given D, V, and N for $\mathcal{J}$, a cost function $Cost_{\mathcal{J}}(\cdot)$ returns the runtime of the job, $T_{\mathcal{J}}$. A spark job often has multiple stages which consist of parallel tasks. For a stage, we consider the longest execution time of a task as the stage completion time. The cost model of a job is described as follows:

$$T_{\mathcal{J}} = Cost_{\mathcal{J}}(D, V, N) = O_{\mathcal{J}} + \sum T_{\mathcal{S}}(D, V, N) \tag{5}$$

where $O_{\mathcal{J}}$ is the overhead of preparing and finishing the stages. The cost function of a stage describes as follows:

$$T_{\mathcal{S}}(D, V, N) = O_{\mathcal{S}}(N) + \lceil \frac{P}{V} \rceil \sum T_{\mathcal{T}}(\frac{D}{P}) \tag{6}$$

where $O_{\mathcal{S}}$ is the overhead of preparing and finishing tasks in a stage. P is the number of partitions, and Spark assigns one vcore to each partition. Here, $\frac{P}{V}$ means that the computing time is inversely linear to vcore [11]. A task also has computing, shuffling steps, and overhead. The cost function of a task describes as follows:

$$T_{\mathcal{T}}(D) = O_{\mathcal{T}} + T_C(D) + T_{\mathcal{SH}}(D) \tag{7}$$

where $O_{\mathcal{T}}$ is the overhead of preparing and finishing the task. $T_{\mathcal{SH}}$ is the time for shuffling. The cost function of computing and shuffling part is described as follows:

$$T_C(D) = \frac{D}{C_{ri}} + D \cdot C_{com} + \frac{D}{C_{wo}} \tag{8}$$

$$T_{\mathcal{SH}}(D) = \frac{D \cdot r_{read}}{C_{rs}} + \frac{D \cdot r_{write}}{C_{ws}} \tag{9}$$

where C_{com}, C_{ri}, C_{wo}, C_{rs}, and C_{ws} are the computation, reading input, writing output, shuffle reading, and shuffle writing speeds respectively, for a task. r_{read} and r_{write} are the ratios of shuffle read and shuffle write in terms of D.

Learning the model. C_{com} for different jobs may vary in models, as the core computation varies in jobs while C_{ri}, C_{wo}, C_{rs}, and C_{ws} are fixed for the model, as we assume the fixed network and disk I/O. We collect simulation logs from the history server using *REST APIs*. From logs we collect T'_C, D', and V', and specifically compute $C_{compute}$ via Equation 7. We collect the simulation logs of input data and shuffle sizes for computing r_{read} and r_{write}.

In Equation 8 and Equation 9, C_{ri}, C_{rs}, C_{ws}, and C_{wo} are calculated from local disk speed, HDFS disk speed, and network speed. They are computed as follows. (i) $C_{ri} = min\{C_{network}, C_{hr}\}$; (ii) $C_{wo} = min\{C_{network}, C_{hw}\}$; (iii) $C_{ws} = min\{C_{network}, C_{ldw}\}$; (iv) $C_{rs} = min\{C_{network}, C_{ldr}\}$. where resource profiles $C_{network}$, C_{hr}, C_{hw}, C_{ldr}, and C_{ldw} are network throughput, HDFS read

throughput, HDFS write throughput, local disk read throughput, and local disk write throughput, respectively. We first collect these parameter values by testing the throughput performance with *Linux dd* command, then properly revise by measuring the performance of some specific stressing workloads (e.g., sort for shuffling throughput). Note that we do not compute these values from the simulation logs, as the sample runs with small amount of input data often do not reach bottleneck throughput.

Minimizing resource time T_R. Recall the resource time $T_R = T_{\mathcal{J}}(V) \times (V + \theta M)$. As memory assurance avoids job failures or deteriorating performance, M is fixed when input data of size D is given. For clarity, we transform the cost model $T_{\mathcal{J}}$ in Equation 5 into $T_{\mathcal{J}}(V) = \beta_0 \frac{1}{V} + \frac{\beta_1}{k} V + \beta_2$, when D is given and $N = \frac{V}{k}$, where k is the number of vcores in one machine. We substitute β_3 with θM, and achieve $T_R(V) = (\beta_0 \frac{1}{V} + \frac{\beta_1}{k} V + \beta_2) \times (V + \beta_3) = \frac{\beta_1}{k} V^2 + (\frac{\beta_1 \beta_3}{k} + \beta_2) V + \frac{\beta_0 \beta_3}{V} + \beta_2 \beta_4 + \beta_0$.

Theoretically, $T_R(V)$ is *unimodal*. With $V = x_0$, which is the root of $T'_R(V) = 0$, we provide the lower bound of T_R as follows:

$$L_{T_R} = \frac{\beta_1}{k} x_0^2 + (\frac{\beta_1 \beta_3}{k} + \beta_2) x_0 + \frac{\beta_0 \beta_3}{x_0} + \beta_2 \beta_3 + \beta_0 \qquad (10)$$

We achieve the lowest resource time when assigning $V = x_0$. It means that if we assign more vcore than x_0, the job performs worse concerning T_R. The reasons are twofolds: (i) after a certain point, the vcore resource is not likely a bottleneck resource, and it will not be fair to use more vcores to reduce the computation time; (ii) N increases as V increases, which leads to higher overhead like communication cost. While in practice, $V = x_0$ may be too small to satisfy user's deadline T_D requirement. We find $V = V_D$ such that $T_{\mathcal{J}} \leq T_D$, i.e., we achieve the lowest resource time, $T_R(V_D)$, under the constraint of deadline T_D.

We propose an *iterative range search* algorithm, i.e., *iterativeRS* in Algorithm 1, which narrows down the range to the (near-) optimal value. We achieve v_{opt} by iterating $RS(\cdot)$ (which finds a small range that includes optimal value), with the worst case $log_d(2|\mathcal{D}|)$ rounds, where $|\mathcal{D}|$ is the total number of configurable values. The iteration stops by meeting a stopping criteria $\lceil \frac{|\mathcal{D}|}{d} \rceil \leq \epsilon$, where d is the distance between two consecutive parameter values for one time search and ϵ is the tolerant error step size.

Algorithm 1: *iterativeRS*

Input: $\mathcal{D}_V$: vcore domain ϵ_e: step size d: distance between two parameter values M: assured memory T_D: deadline
Output: V_k: Recommended vcore T_m: (close-to-)optimal metric value

```
v_k, T_m ← ∞ ;                                   // Initial v_k and metric
while ⌈|D_V|/d⌉ ≤ ε_e do
    V ← Grading(D_V, d)
    p ← d/|D| ; v_k = arg min_{V∈D_V} f(v)
    foreach v ∈ V do
        t ← Cost(v) ;                            // refer to Equation 5
        T'_m ← computeMetric(v, M, t);           // refer to Equation 1
        if T'_m < T_m then
            T_m ← T'_m ; v_k ← v
    D'_v ← RS(D'_v, d, v_k) ;                    // narrow down the space
    D_v ← D'_v
return (V_k, T_m) ;
```

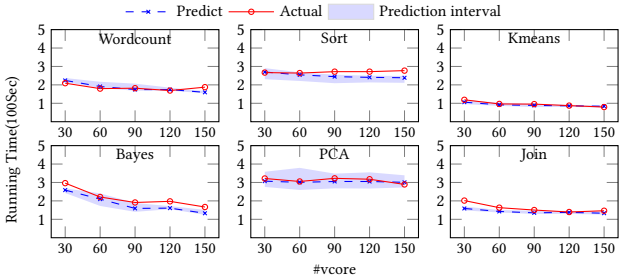

Figure 1: Performance prediction of Spark workloads. Predicted time (dashed curve), actual time (solid curve) and min-max prediction interval.

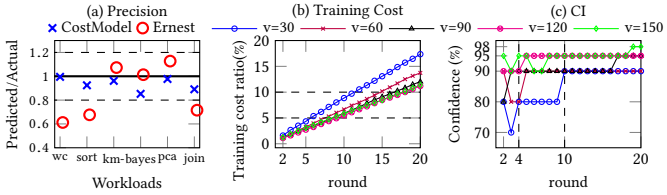

Figure 2: (a) Average precision of execution time predictions. The closer to 1, the better. Case of *Sort*: (b) training cost ratio and (c) confidence interval of the training rounds.

4 EVALUATION

Experimental setup. We made empirical experiments with HiBench workloads in a YARN Spark cluster, which has 10 nodes, each with 32*GB* RAM and 2 (4 cores) Xeon E5540 2.53GHz CPUs. The cores are using *hyperthreading*, so there is a total of 16 threads (vcores) in one node. We conducted all the experiments with *CGroups* enforcement on Yarn. Thus the performance of vcore and memory allocations is isolated. Table 1 describes benchmark workloads of 3 categories with their input data sizes and deadlines. The maximum possible resource configuration is (150v,150g). The number of files is 30 (providing us to sample $p = 1/30$ or $p = 1/15$ of the data), and each file is split into 5 partitions. The cost of sampling data is ignorable to the real runs.

Memory assurance. For simulation, we deployed 3 sample probabilities, which are $p = 1/30$, $p = 1/15$, and $p = 1/10$. For Kmeans, we pick 0.6*GB*, 1.2*GB*, and 1.8*GB*, respectively for these three probabilities, as the proper memory values assured by the elbow points. The predicted recommended memory of the workloads are shown in Table 1.

Performance prediction. Figure 1 shows the prediction time vs. actual runtime of workloads with varied vcore numbers. The predictions are quite satisfactory for most workloads, as our cost model can capture jobs' characteristics from sample simulations. For example, the difference between the prediction time and actual time for Wordcount, Kmeans, and PCA workloads are smaller than 5% with various numbers of vcores. The prediction interval widths of Wordcount, Sort, Kmeans, and Join are short, which in turn indicates small training costs, as we will show in Table 1.

Prediction accuracy. We define the precision metric, i.e., accuracy, as the predicted runtime divided by the actual runtime. Figure 2(a) shows the average precision of the benchmark workloads. The average prediction error for cost model (Equation 5) is under 15% for each workload, and the overall average error is under 7%. However, such error does not impact our decision on selecting

Table 1: Evaluation of Spark workloads in terms of resource time and training cost ratio (last two column) for parameter recommendation.

Type	Workload	Data size	deadline	Total Rec	T_R speedup	CI=80%	CI=90%
Micro	Wordcount	91.8G	200s	60v18g	217%	4.11%	4.47%
	Sort	9.55G	300s	30v18g	492%	1.99%	2.37%
ML	Kmeans	11.2G	100s	60v36g	229%	5.89%	6.51%
	Bayesian	14.0G	200s	90v54g	161%	7.91%	16.89%
	PCA	30.7M	350s	30v18g	499%	9.36%	41.80%
SQL	Join	17.3G	200s	30v18g	360%	4.99%	5.22%

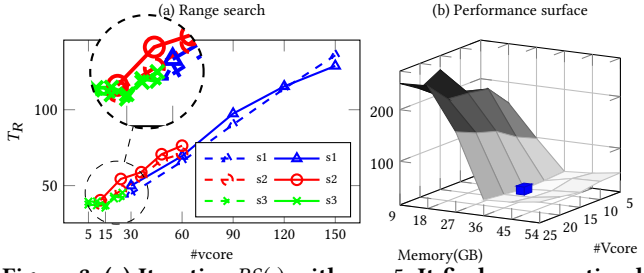

Figure 3: (a) Iterative $RS(\cdot)$ with $\epsilon = 5$. It finds near optimal value $v = 15$ in three rounds for Kmeans. s1, s2, and s3 stand for first, second, and third range search, respectively. (b) Kmeans: evaluation of recommendation. (15 vcores, 36 GB) is the recommended value and actual optimal value among options concerning resource time.

the proper values among the vcore options. We further compare our method with Ernest [11]. Our cost model performs as good as Ernest in predicting vcores for ML workloads. We perform better in Mirco and SQL workloads, as we predict the performance of these workloads by considering the network and disk bottlenecks.

Training cost. Table 1 shows the training cost for all the considered workloads. When given the confident interval (CI) 80%, the accumulated training cost of simulation is less than 10% for all the workloads. The performance time of LR deviates a lot due to the skewness of data distribution and the subtle system performance changes. Likewise, with 90% CI, the accumulated cost of simulation is less than 20%, except for PCA. Overall, the average accumulated simulation or training costs are 5.71% and 12.88% in terms of one round of actual runs for CI=80% and CI=90%, respectively. Specifically, Figure 2(b,c) show the training cost (described in Equation 4) of each performance prediction and confidence interval for the Sort workload. Given a confidence interval of 80% as the stopping criteria, it requires at least four rounds of training, which means the training cost will be less than 5% of the actual run. Similarly, less than 10% of the training cost is required if we set the confidence interval to 90%.

Parameter recommendation. Table 1 shows the recommendation results. We compute the changes in resource time T_R by dividing the runtime for the maximum resource configuration with the runtime for the recommended configuration. We achieve up to 499% T_R speedup compared to the baseline configurations for the workloads. This not only satisfies the user's deadline requirements but also achieves significant resource time T_R gain.

We take Kmeans as an example to further illustrate the performance of *iterativeRS*, i.e., Algorithm 1. Given the stopping step size ϵ=5, size of available vcores D=150, and values distance d=5. Figure 3 illustrates the range search steps for the Kmeans. With the initial $\mathcal{D}_V$ ranging in $[0, 150]$, ϵ_e=5, and d=5, we grade $\mathcal{D}_V$ into V=$\{0, 30, 60, 90, 120, 150\}$. We set T_R to a large number for v=0. From the first range search, we obtain v=30 as the parameter value with the minimum T_R using Equation 2. Since $\lceil \frac{D=150}{d=5} \rceil$=30 > ϵ=5, we continue *iterativeRS* with new $\mathcal{D}_V$ ranging in $[0, 60]$. Likewise, we obtain that v=12 is the value with the minimum T_R. Again, since $\frac{60}{5} > \epsilon$, we continue the search with new $\mathcal{D}_V$ ranging in $[0, 24]$. We get V=$\{0, 5, 10, 15, 20, 25\}$ by rounding up. We find $v = 15$ satisfying the minimum T_R among all the options. Now, since $\lceil \frac{24}{5} \rceil \leq \epsilon$, we stop searching and return v=15. In Table 1, we have shown the assured memory for Kmeans workload is 36G. Hence, the recommended configuration (v=15, m=36G) with ϵ=5 is optimal among the options shown in Figure 3(b) from the actual runs.

5 CONCLUSION AND FUTURE WORK

This short paper introduces a sampling and simulation framework that predicts and recommends the total amount of the vcore and memory resources toward cost-effectiveness of Spark Big data jobs. The empirical experiments show the efficiency and effectiveness of our proposed algorithms. As our future work, we would like to improve the searching algorithm and provide more insights of the models.

6 ACKNOWLEDGMENTS

This work is partially supported by Huawei HIRP open project (project No. HO2016050002AB), Academy of Finland (project No. 310321), and Crowdsourced Battery Optimization AI for a Connected World (CBAI) (project No. 319017).

REFERENCES

[1] K. Binder. Monte carlo simulations in statistical physics. In *Encyclopedia of Complexity and Systems Science*, pages 5667–5677. Springer, 2009.
[2] K. Chen, J. Powers, S. Guo, and F. Tian. CRESP: towards optimal resource provisioning for mapreduce computing in public clouds. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1403–1412, 2014.
[3] Y. Chen, X. Qin, H. Bian, J. Chen, Z. Dong, X. Du, Y. Gao, D. Liu, J. Lu, and H. Zhang. A study of sql-on-hadoop systems. In *BPOE*, pages 154–166, 2014.
[4] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres. Dynamic configuration of partitioning in spark applications. *IEEE Trans. Parallel Distrib. Syst.*, 28(7):1891–1904, 2017.
[5] Á. B. Hernández, M. S. Perez, S. Gupta, and V. Muntés-Mulero. Using machine learning to optimize parallelism in big data applications. *Future Generation Computer Systems*, 86:1076–1092, 2018.
[6] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDE Workshops*, pages 41–51. IEEE Computer Society, 2010.
[7] D. J. Ketchen and C. L. Shook. The application of cluster analysis in strategic management research: an analysis and critique. *Strategic management journal*, 17(6):441–458, 1996.
[8] J. Lu, Y. Chen, H. Herodotou, and S. Babu. Speedup your analytics: Automatic parameter tuning for databases and big data systems. *PVLDB*, 12(21):1970–1973, 2019.
[9] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang. Mrtuner: A toolkit to enable holistic optimization for mapreduce jobs. *PVLDB*, 7(13):1319–1330, 2014.
[10] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD Conference*, pages 953–966. ACM, 2008.
[11] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, pages 363–378. USENIX Association, 2016.
[12] G. Wang, J. Xu, and B. He. A novel method for tuning configuration parameters of spark based on machine learning. In *HPCC/SmartCity/DSS*, pages 586–593. IEEE Computer Society, 2016.
[13] K. Wang and M. M. H. Khan. Performance prediction for apache spark platform. In *HPCC/CSS/ICESS*, pages 166–173. IEEE, 2015.
[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28. USENIX Association, 2012.